libcga Reference Manual

Generated by Doxygen 1.3.7

Mon Sep 20 13:48:51 2004

Contents

1 Libcga Documentation **1**

2 libcga Module Index **3**

2.1 libcga Modules 3

3 libcga Data Structure Index **5**

3.1 libcga Data Structures 5

4 libcga File Index **7**

4.1 libcga File List 7

5 libcga Page Index **9**

5.1 libcga Related Pages 9

6 libcga Module Documentation **11**

6.1 Core multivector manipulation functions 11

6.2 Rotor interpolation functions for CGA 18

6.3 Standard functions for CGA 20

7 libcga Data Structure Documentation **23**

7.1 multiv_s Struct Reference 23

7.2 plugin_s Struct Reference 25

8 libcga File Documentation **27**

8.1 cga.h File Reference 27

8.2 cga_rotor_interpolate.h File Reference 29

8.3 cga_stdfunc.h File Reference 30

9 libcga Page Documentation **31**

9.1 Examples of Libcga Use 31

9.2 A Simple Example 32

9.3 Introduction to Geometric Algebra 34
9.4 Grade Tracking 39

Chapter 1

Libcga Documentation

Introduction

libcga is a C library designed to allow efficient implementation of algorithms which use **Geometric Algebra**(p. 34). The library has the following features:

1. It uses a modular, plugin-based approach to different algebras. This enables both extension of the library to new algebras and optimisations in existing algebras to be introduced without re-compiling programs which use the library. Also 3rd party algebra implementations may be included.
2. The default algebra implementations utilise the **Grade Tracking**(p. 39) optimisation to provide fast, efficient product calculation.
3. Various visualisation routines are provided to allow rapid display of results in a variety of algebras and geometries (currently 3D Euclidean, 2D Hyperbolic and 3D Hyperbolic).

libcga is developed under `Linux`, but is designed to be portable. As a result, it runs on most other Unix flavors as well. Furthermore, compilation under `Cygwin` is possible on Windows systems.

This manual is divided into two parts, each of which is divided into several sections.

The first part forms a user manual:

- **Introduction to Geometric Algebra**(p. 34) provides a brief overview of the concepts in Geometric Algebra.
- **Examples of Libcga Use**(p. 31) provides some simple examples of using libcga.

The second part forms a reference manual:

- **Core multivector manipulation functions**(p. 11) describes the core multivector manipulation routines.
- **Standard functions for CGA**(p. 20) describes some standard functions for working with multivectors
- **Rotor interpolation functions for CGA**(p. 18) describes some functions useful for interpolating rotors.
- drawing describes the functions available for visualising operations and multivectors in 3D Euclidean CGA.

- drawing_2dh describes the functions available for visualising operations and multivectors in 2D Hyperbolic CGA.
- drawing_3dh describes the functions available for visualising operations and multivectors in 3D Hyperbolic CGA.

libcga license

Copyright ©2001-2003 by `Richard Wareham`. Portions of the test programs Copyright ©1999 Paul Rademacher.

Permission to use, copy, modify, and distribute this software and its documentation under the terms of the GNU General Public License is hereby granted. No representations are made about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. See the `GNU General Public License` for more details.

Future work

Acknowledgements

Thanks go to:

- Add acknowledgements

Chapter 2

libcga Module Index

2.1 libcga Modules

Here is a list of all modules:

Core multivector manipulation functions 11
Rotor interpolation functions for CGA 18
Standard functions for CGA 20

Chapter 3

libcga Data Structure Index

3.1 libcga Data Structures

Here are the data structures with brief descriptions:

multiv_s 23
plugin_s 25

Chapter 4

libcga File Index

4.1 libcga File List

Here is a list of all documented files with brief descriptions:

cga.h 27
cga_internal.h . . . . . ??
cga_rotor_interpolate.h 29
cga_stdfunc.h 30

Chapter 5

libcga Page Index

5.1 libcga Related Pages

Here is a list of all related documentation pages:

Examples of Libcga Use 31
A Simple Example 32
Introduction to Geometric Algebra 34
Grade Tracking 39

Chapter 6

libcga Module Documentation

6.1 Core multivector manipulation functions

Data Structures

- struct **multiv_s**

Defines

- #define **CGA_TYPE** float
- #define **CGA_OPT_GRADETRACKING** 0x001

Typedefs

- typedef **multiv_s** * **multiv_t**

Functions

- CGA_SCOPE int **cga_init** ()
- CGA_SCOPE void **cga_finalise** ()
- CGA_SCOPE int **cga_load_algebra** (char *name)
- CGA_SCOPE **multiv_t** **cga_new_multiv** ()
- CGA_SCOPE void **cga_free_multiv** (**multiv_t** mv)
- CGA_SCOPE **multiv_t** **cga_stdmv** (const char *specifier)
- CGA_SCOPE CGA_TYPE * **cga_get_element** (**multiv_t** mv, int grade, int id)
- CGA_SCOPE void **cga_dump** (**multiv_t** mv)
- CGA_SCOPE void **cga_reverse** (**multiv_t** src, **multiv_t** dest)
- CGA_SCOPE void **cga_add** (**multiv_t** a, **multiv_t** b, **multiv_t** c)
- CGA_SCOPE void **cga_subtract** (**multiv_t** a, **multiv_t** b, **multiv_t** c)
- CGA_SCOPE void **cga_outer** (**multiv_t** a, **multiv_t** b, **multiv_t** c)
- CGA_SCOPE void **cga_inner** (**multiv_t** a, **multiv_t** b, **multiv_t** c)
- CGA_SCOPE void **cga_geometric** (**multiv_t** a, **multiv_t** b, **multiv_t** c)
- CGA_SCOPE void **cga_meet** (**multiv_t** a, **multiv_t** b, **multiv_t** c)
- CGA_SCOPE void **cga_cpy** (**multiv_t** to, **multiv_t** from)

- CGA_SCOPE void **cga_extract_grade** (int n, **multiv_t** a, **multiv_t** b)
- CGA_SCOPE int **cga_factorise_bivector** (**multiv_t** mv, **multiv_t** a, **multiv_t** b)
- CGA_SCOPE CGA_TYPE **cga_mag2** (**multiv_t** a)
- CGA_SCOPE void **cga_dual** (**multiv_t** a, **multiv_t** b)
- CGA_SCOPE void **cga_scale** (CGA_TYPE beta, **multiv_t** mv)
- CGA_SCOPE void **cga_reset_flops** ()
- CGA_SCOPE unsigned long **cga_flops** ()
- CGA_SCOPE void **cga_set_optimisations** (int opt_flags)
- CGA_SCOPE int **cga_optimisation_flags** ()
- CGA_SCOPE void **cga_make_zero** (**multiv_t** mv)

6.1.1 Detailed Description

These are the functions that nearly all programs using libcga will need to use. They provide the basic products and multivector manipulations that you will need. All other parts of the library use these functions.

6.1.2 Define Documentation

6.1.2.1 #define CGA_TYPE float

The type used to represent multivector components, etc.

6.1.3 Typedef Documentation

6.1.3.1 typedef multiv_s* multiv_t

A pointer to a **multiv_s**(p. 23) structure.

See also:
The **multiv_s**(p. 23) structure.

6.1.4 Function Documentation

6.1.4.1 CGA_SCOPE int cga_init ()

Initialise the CGA library and load the default plugin ('5dcga').

Returns:
0 if there was an error starting, non-zero otherwise.

See also:
cga_finalise()(p. 14), **cga_load_algebra()**(p. 14)

6.1.4.2 CGA_SCOPE void cga_finalise ()

Finalise the CGA library (unloading any loaded plugins).

See also:
cga_init()(p. 14)

6.1.4.3 CGA_SCOPE int cga_load_algebra (char * *name*)

Loads a particular algebra by name. Currently the following plugins are included by default:

3dcga Conformal extension of 3D Euclidean space, signature is (4,1)

2dcga Conformal extension of 2D Euclidean space, signature is (3,1)

3ddesit Conformal extension of 3D deSitter space, signature is ++-+-

2ddesit Conformal extension of 2D deSitter space, signature is +-+-

5d 5D Euclidean space, signature (5,0)

4d 4D Euclidean space, signature (4,0)

3d 3D Euclidean space, signature (3,0)

5d 5D Euclidean space, signature (5,0)

Returns:
0 on failure, non-zero on success

6.1.4.4 CGA_SCOPE multiv_t cga_new_multiv ()

Creates a multivector and initialises it to zero.

Returns:
A pointer to the new multivector which must be freed with **cga_free_multiv()**(p. 15) or NULL on error.

See also:
cga_free_multiv()(p. 15)

6.1.4.5 CGA_SCOPE void cga_free_multiv (multiv_t *mv*)

Frees the resources associated with a multivector previously initialised by **cga_new_multiv()**(p. 14)

Parameters:
mv The multivector whose resources should be freed.

See also:
cga_new_multiv()(p. 14)

6.1.4.6 CGA_SCOPE multiv_t cga_stdmv (const char * *specifier*)

Creates and returns a 'standard' multivector. For example, 'e0' gives the unit scalar, 'e12' gives $e_1 \wedge e_2$ and 'I' gives the pseudoscalar.

Parameters:
specifier The name of the standard multivector to return.

Returns:
The created multivector. This must be freed with **cga_free_multiv()**(p. 15) when it is no-longer needed.

6.1.4.7 CGA_SCOPE CGA_TYPE* cga_get_element (multiv_t *mv*, int *grade*, int *id*)

Get a pointer to the component with specified grade and id. For example the e_{124} component has grade 3, id 124.

Returns:
A pointer to the specified component or NULL if the component grade/id was invalid.

6.1.4.8 CGA_SCOPE void cga_dump (multiv_t *mv*)

Dumps the contents of a multivector to stdout.

Parameters:
mv The multivector to dump.

6.1.4.9 CGA_SCOPE void cga_reverse (multiv_t *src*, multiv_t *dest*)

Find the reverse of a multivector.

Parameters:
src The multivector to find the reverse of.

dest The multivector to write the result into.

6.1.4.10 CGA_SCOPE void cga_add (multiv_t *a*, multiv_t *b*, multiv_t *c*)

Adds two multivectors together.

Parameters:
a The first multivector.

b The second multivector.

c The multivector that the result of $a + b$ is written into.

6.1.4.11 CGA_SCOPE void cga_subtract (multiv_t *a*, multiv_t *b*, multiv_t *c*)

Subtracts one multivector from another.

Parameters:
a The first multivector.

b The second multivector.

c The multivector that the result of $a - b$ is written into.

6.1.4.12 CGA_SCOPE void cga_outer (multiv_t *a*, multiv_t *b*, multiv_t *c*)

Find the outer (aka 'wedge') product of two multivectors.

Parameters:

a The first multivector.

b The second multivector.

c The multivector that the result of $a \wedge b$ is written into.

6.1.4.13 CGA_SCOPE void cga_inner (multiv_t *a*, multiv_t *b*, multiv_t *c*)

Find the inner (aka 'dot') product of two multivectors.

Parameters:

a The first multivector.

b The second multivector.

c The multivector that the result of $a \cdot b$ is written into.

6.1.4.14 CGA_SCOPE void cga_geometric (multiv_t *a*, multiv_t *b*, multiv_t *c*)

Find the geometric (aka 'Clifford') product of two multivectors.

Parameters:

a The first multivector.

b The second multivector.

c The multivector that the result of ab is written into.

6.1.4.15 CGA_SCOPE void cga_meet (multiv_t *a*, multiv_t *b*, multiv_t *c*)

Find the meet of two multivectors.

Parameters:

a The first multivector.

b The second multivector.

c The multivector that the result of $a \vee b$ is written into.

6.1.4.16 CGA_SCOPE void cga_cpy (multiv_t *to*, multiv_t *from*)

Copies one multivector inot another.

Parameters:

to The multivector to copy into.

from The multivector to copt from.

6.1.4.17 CGA_SCOPE void cga_extract_grade (int *n*, multiv_t *a*, multiv_t *b*)

Sets all but grade 'n' elements to zero.

Parameters:

n The grade to preserve.

a The source multivector.

b The multivector to write the result into.

6.1.4.18 CGA_SCOPE int cga_factorise_bivector (multiv_t *mv*, multiv_t *a*, multiv_t *b*)

Factorises a bivector $A \wedge B$ into two null-vectors A and B.

Parameters:

mv The multivector to factorise.

a One multivector to write the result into.

b The other multivector to write the result into.

Returns:

0 if this cannot be done, non-zero otherwise.

6.1.4.19 CGA_SCOPE CGA_TYPE cga_mag2 (multiv_t *a*)

Returns the magnitude SQUARED of the multivector, i.e. returns a^2

6.1.4.20 CGA_SCOPE void cga_dual (multiv_t *a*, multiv_t *b*)

Sets b to the dual of a.

6.1.4.21 CGA_SCOPE void cga_scale (CGA_TYPE *beta*, multiv_t *mv*)

Scales mv by a scalar beta.

6.1.4.22 CGA_SCOPE void cga_reset_flops ()

Resets the global FLOP count to zero

6.1.4.23 CGA_SCOPE unsigned long cga_flops ()

Returns the global FLOP count

6.1.4.24 CGA_SCOPE void cga_set_optimisations (int *opt_flags*)

Sets the optimisation flags.

Note:

This is mainly intended for debugging and benchmarking the library.

6.1.4.25 CGA_SCOPE int cga_optimisation_flags ()

Gets the optimisation flags

Note:
This is mainly intended for debugging and benchmarking the library.

6.1.4.26 CGA_SCOPE void cga_make_zero (multiv_t *mv*)

Sets all components and the grade mask of the passed multivector to zero.

Parameters:
mv The multivector to set to zero.

6.2 Rotor interpolation functions for CGA

Functions

- CGA_SCOPE void **cga_matrix_to_action** (float *matrix, **multiv_t** plane, float *angle)
- CGA_SCOPE void **cga_matrix_to_rotor** (float *m, **multiv_t** rotor)
- CGA_SCOPE void **cga_rotor_to_matrix** (**multiv_t** rotor, float *m)
- CGA_SCOPE void **cga_rotor_log** (**multiv_t** rotor, **multiv_t** log)
- CGA_SCOPE void **cga_rotor_exp** (**multiv_t** log, **multiv_t** rotor)

6.2.1 Detailed Description

These functions implement the method of rotor interpolation by logarithm-like functions.

6.2.2 Function Documentation

6.2.2.1 CGA_SCOPE void cga_matrix_to_action (float * *matrix*, multiv_t *plane*, float * *angle*)

This function is still buggy!

Converts a 4x4 rotation matrix into the corresponding rotation plane (expressed as a bivector parallel to it) and the angle of rotation within the plane. The matrix is passed as an array of floats as in glMultMatrix(3) and friends. The elements of the array m are arranged in the matrix thus

$$\begin{pmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{pmatrix}$$

Parameters:

matrix The matrix to convert into a rotor.

plane The multiv_t which gets written with the plane.

angle Pointer to a float to write the angle into.

See also:

cga_rotor_to_matrix()(p. ??)

6.2.2.2 CGA_SCOPE void cga_matrix_to_rotor (float * *m*, multiv_t *rotor*)

This function is still buggy!

Converts a 4x4 rotation matrix into the corresponding rotor (assuming we are using the 3dcga algebra). The matrix is passed as an array of floats as in glMultMatrix(3) and friends. The elements of the array m are arranged in the matrix thus

$$\begin{pmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{pmatrix}$$

Parameters:

m The matrix to convert into a rotor.

rotor The multiv_t which gets written with the rotor.

See also:

cga_rotor_to_matrix()(p. ??)

6.2.2.3 CGA_SCOPE void cga_rotor_to_matrix (multiv_t *rotor*, float * *m*)

Like **cga_matrix_to_rotor()**(p. ??) but works in reverse.

Parameters:

rotor The rotor to convert into a 4x4 matrix.

m The array containing the matrix elements to write into.

See also:

cga_matrix_to_rotor()(p. ??)

6.2.2.4 CGA_SCOPE void cga_rotor_log (multiv_t *rotor*, multiv_t *log*)

Takes the pseudo-logarithm of the rotor passed.

Parameters:

rotor The rotor to work on.

log The multiv_t to write the log into.

See also:

cga_rotor_exp()(p. ??)

6.2.2.5 CGA_SCOPE void cga_rotor_exp (multiv_t *log*, multiv_t *rotor*)

The reverse of **cga_rotor_log()**(p. ??), forms the original rotor from the pseudo-logarithm.

Parameters:

log The pseudo-logarithm of the rotor.

rotor The multiv_t to write the reultant rotor into.

See also:

cga_rotor_log()(p. ??)

6.3 Standard functions for CGA

Defines

- #define **cga_assign_vector**(mv, components, num_components)

Functions

- CGA_SCOPE void **cga_map** (**multiv_t** x)
- CGA_SCOPE void **cga_imap** (**multiv_t** x)
- CGA_SCOPE void **cga_maph** (**multiv_t** x)
- CGA_SCOPE void **cga_imaph** (**multiv_t** x)
- CGA_SCOPE void **cga_rotor** (CGA_TYPE rad, **multiv_t** plane, **multiv_t** rot)
- CGA_SCOPE void **cga_translator** (**multiv_t** x, **multiv_t** T)
- CGA_SCOPE void **cga_dilator** (CGA_TYPE rho, **multiv_t** D)
- CGA_SCOPE void **cga_rotorh** (CGA_TYPE rad, **multiv_t** plane, **multiv_t** rot)
- CGA_SCOPE void **cga_translatorh** (**multiv_t** x, **multiv_t** T)
- CGA_SCOPE void **cga_sphere** (**multiv_t** C, CGA_TYPE radius, **multiv_t** sigma)

6.3.1 Detailed Description

These functions implement the various n-d vector to $(n+2)$-d null-vector mappings that can be performed along with appropriate functions to calculate rotors.

6.3.2 Define Documentation

6.3.2.1 #define cga_assign_vector(mv, components, num_components)

Value:

```
cga_make_zero(mv); mv->grade_mask |= (1 << 1); \
 memcpy(cga_get_element(mv, 1, 1), components, num_components * sizeof(CGA_TYPE))
```

Assigns a multivector the vector with components passed. This is implemented as a macro and is designed as a 'quick-and-dirty' way of doing what should really be done properly.

Parameters:

mv The multivector to assign the vector to.

components An array of components.

num_components The number of components. Note that the behaviour when trying to assign more components than the multivector can take is un-defined.

6.3.3 Function Documentation

6.3.3.1 CGA_SCOPE void cga_map (multiv_t *x*)

Map the Euclidean point x into a null-vector using the Euclidean mapping $x \mapsto \frac{1}{2\lambda}(x^2 n + 2\lambda x - \lambda^2 \bar{n})$ with $\lambda = 1$.

Parameters:

x The vector to be mapped. This vector is directly modified so that after the function call it contains the mapped null-vector.

6.3.3.2 CGA_SCOPE void cga_imap (multiv_t *x*)

Map the null-vector x into a point using the inverse of the Euclidean mapping $x \mapsto \frac{1}{2\lambda}(x^2 n + 2\lambda x - \lambda^2 \bar{n})$ with $\lambda = 1$.

Parameters:

x The null-vector to be inverse-mapped. This vector is directly modified so that after the function call it contains the original point.

6.3.3.3 CGA_SCOPE void cga_maph (multiv_t *x*)

Map the Euclidean point x which must be within the unit-disc into a null-vector using the non-Euclidean (hyperbolic) mapping $x \mapsto \frac{1}{\lambda^2 - x^2}(x^2 n + 2\lambda x - \lambda^2 \bar{n})$ with $\lambda = 1$.

Parameters:

x The vector to be mapped. This vector is directly modified so that after the function call it contains the mapped null-vector.

6.3.3.4 CGA_SCOPE void cga_imaph (multiv_t *x*)

Map the null-vector x into a point within the unit-circle using the inverse of the non-Euclidean (hyperbolic) mapping $x \mapsto \frac{1}{\lambda^2 - x^2}(x^2 n + 2\lambda x - \lambda^2 \bar{n})$ with $\lambda = 1$.

Parameters:

x The null-vector to be inverse-mapped. This vector is directly modified so that after the function call it contains the original point.

6.3.3.5 CGA_SCOPE void cga_rotor (CGA_TYPE *rad*, multiv_t *plane*, multiv_t *rot*)

Form a Euclidean rotation rotor. The rotor will rotate in a specified plane by a specified amount.

Parameters:

rad The angle, in radians, to rotate by.

plane The plane to rotate in (e.g. e_{12} to rotate in plane spanned by e_1 and e_2).

rot This multivector will be over-written with the rotor.

6.3.3.6 CGA_SCOPE void cga_translator (multiv_t *x*, multiv_t *T*)

Form a Euclidean translation rotor.

Parameters:

x The (Euclidean) point that the translator should map the origin to.

T This multivector will be over-written with the rotor.

6.3.3.7 CGA_SCOPE void cga_dilator (CGA_TYPE *rho*, multiv_t *D*)

Form a Euclidean dilator rotor. FIXME: This function is not yet generalised beyond the "3dcga" algebra.

Parameters:

rho The dilation factor factor.

D This multivector will be over-written with the rotor.

6.3.3.8 CGA_SCOPE void cga_rotorh (CGA_TYPE *rad*, multiv_t *plane*, multiv_t *rot*)

Form a hyperbolic rotation rotor. The rotor will rotate in a specified plane by a specified amount.

Parameters:

rad The angle, in radians, to rotate by.

plane The plane to rotate in (e.g. e_{12} to rotate in plane spanned by e_1 and e_2).

rot This multivector will be over-written with the rotor.

6.3.3.9 CGA_SCOPE void cga_translatorh (multiv_t *x*, multiv_t *T*)

Form a hyperbolic translation rotor.

Parameters:

x The (Euclidean) point within the unit-circle that the translator should map the origin to.

T This multivector will be over-written with the rotor.

6.3.3.10 CGA_SCOPE void cga_sphere (multiv_t *C*, CGA_TYPE *radius*, multiv_t *sigma*)

Creates a multivector with a spherical INNER null-space, i.e. the dual of a sphere, with a given centre and radius.

Parameters:

C The null-vector representation of the Euclidean point giving the centre.

radius The radius.

sigma This multivector will be overwritten with the sphere.

Chapter 7

libcga Data Structure Documentation

7.1 multiv_s Struct Reference

`#include <cga.h>`

Data Fields

- uint32_t **max_grade**
- uint32_t **grade_mask**
- CGA_TYPE * **components**
- uint32_t **num_components**
- unsigned char **dimension**

7.1.1 Detailed Description

The multiv_t type encapsulates a generic multivector. A multivector is specified in terms of a set of components proportional to the basis elements $\{e_1, e_2, ..., e_{12}, e_{23}, ..., e_{ijk...}\}$ and a 'grade-mask' which shows which grades are present.

7.1.2 Field Documentation

7.1.2.1 uint32_t multiv_s::max_grade

Maximum grade of any element

7.1.2.2 uint32_t multiv_s::grade_mask

If bit 'n' set, multivector has component of grade 'n'

7.1.2.3 CGA_TYPE* multiv_s::components

A pointer to an num_components length array of components

7.1.2.4 uint32_t multiv_s::num_components

Number of components

7.1.2.5 unsigned char multiv_s::dimension

Dimension of space

The documentation for this struct was generated from the following file:

- **cga.h**

7.2 plugin_s Struct Reference

`#include <cga_internal.h>`

Data Fields

- cga_prod_f **geometric_prod**
- cga_prod_f **outer_prod**
- cga_prod_f **hestenes_prod**
- cga_prod_f **dot_prod**
- cga_prod_f **scalar_prod**
- cga_reverse_f **reverse**
- cga_project_f **project_to_grade**
- cga_new_f **new_mv**
- cga_element_accessor_f **element_accessor**
- cga_stdmv_f **stdmv**
- cga_dump_f **dump**

7.2.1 Detailed Description

Internal to libcga.

The documentation for this struct was generated from the following file:

- cga_internal.h

Chapter 8

libcga File Documentation

8.1 cga.h File Reference

Data Structures

- struct **multiv_s**

Defines

- #define **CGA_SCOPE** extern
- #define **CGA_TYPE** float
- #define **CGA_OPT_GRADETRACKING** 0x001

Typedefs

- typedef unsigned int **uint32_t**
- typedef **multiv_s** * **multiv_t**

Functions

- CGA_SCOPE int **cga_init** ()
- CGA_SCOPE void **cga_finalise** ()
- CGA_SCOPE int **cga_load_algebra** (char *name)
- CGA_SCOPE **multiv_t** **cga_new_multiv** ()
- CGA_SCOPE void **cga_free_multiv** (**multiv_t** mv)
- CGA_SCOPE **multiv_t** **cga_stdmv** (const char *specifier)
- CGA_SCOPE CGA_TYPE * **cga_get_element** (**multiv_t** mv, int grade, int id)
- CGA_SCOPE void **cga_dump** (**multiv_t** mv)
- CGA_SCOPE void **cga_reverse** (**multiv_t** src, **multiv_t** dest)
- CGA_SCOPE void **cga_add** (**multiv_t** a, **multiv_t** b, **multiv_t** c)
- CGA_SCOPE void **cga_subtract** (**multiv_t** a, **multiv_t** b, **multiv_t** c)
- CGA_SCOPE void **cga_outer** (**multiv_t** a, **multiv_t** b, **multiv_t** c)
- CGA_SCOPE void **cga_inner** (**multiv_t** a, **multiv_t** b, **multiv_t** c)
- CGA_SCOPE void **cga_geometric** (**multiv_t** a, **multiv_t** b, **multiv_t** c)

- CGA_SCOPE void **cga_meet** (**multiv_t** a, **multiv_t** b, **multiv_t** c)
- CGA_SCOPE void **cga_cpy** (**multiv_t** to, **multiv_t** from)
- CGA_SCOPE void **cga_extract_grade** (int n, **multiv_t** a, **multiv_t** b)
- CGA_SCOPE int **cga_factorise_bivector** (**multiv_t** mv, **multiv_t** a, **multiv_t** b)
- CGA_SCOPE CGA_TYPE **cga_mag2** (**multiv_t** a)
- CGA_SCOPE void **cga_dual** (**multiv_t** a, **multiv_t** b)
- CGA_SCOPE void **cga_scale** (CGA_TYPE beta, **multiv_t** mv)
- CGA_SCOPE void **cga_reset_flops** ()
- CGA_SCOPE unsigned long **cga_flops** ()
- CGA_SCOPE void **cga_set_optimisations** (int opt_flags)
- CGA_SCOPE int **cga_optimisation_flags** ()
- CGA_SCOPE void **cga_make_zero** (**multiv_t** mv)

8.1.1 Detailed Description

Core multivector manipulation functions

8.2 cga_rotor_interpolate.h File Reference

Defines

- #define **CGA_SCOPE** extern

Functions

- CGA_SCOPE void **cga_matrix_to_action** (float *matrix, **multiv_t** plane, float *angle)
- CGA_SCOPE void **cga_matrix_to_rotor** (float *m, **multiv_t** rotor)
- CGA_SCOPE void **cga_rotor_to_matrix** (**multiv_t** rotor, float *m)
- CGA_SCOPE void **cga_rotor_log** (**multiv_t** rotor, **multiv_t** log)
- CGA_SCOPE void **cga_rotor_exp** (**multiv_t** log, **multiv_t** rotor)

8.2.1 Detailed Description

Rotor interpolation functions for CGA

8.3 cga_stdfunc.h File Reference

```
#include <string.h>
```

Defines

- #define **CGA_SCOPE** extern
- #define **cga_assign_vector**(mv, components, num_components)

Functions

- CGA_SCOPE void **cga_map** (**multiv_t** x)
- CGA_SCOPE void **cga_imap** (**multiv_t** x)
- CGA_SCOPE void **cga_maph** (**multiv_t** x)
- CGA_SCOPE void **cga_imaph** (**multiv_t** x)
- CGA_SCOPE void **cga_rotor** (CGA_TYPE rad, **multiv_t** plane, **multiv_t** rot)
- CGA_SCOPE void **cga_translator** (**multiv_t** x, **multiv_t** T)
- CGA_SCOPE void **cga_dilator** (CGA_TYPE rho, **multiv_t** D)
- CGA_SCOPE void **cga_rotorh** (CGA_TYPE rad, **multiv_t** plane, **multiv_t** rot)
- CGA_SCOPE void **cga_translatorh** (**multiv_t** x, **multiv_t** T)
- CGA_SCOPE void **cga_sphere** (**multiv_t** C, CGA_TYPE radius, **multiv_t** sigma)

8.3.1 Detailed Description

Standard functions for CGA.

Chapter 9

libcga Page Documentation

9.1 Examples of Libcga Use

The following sections contain simple example programs to illustrate features of libcga.

- **A Simple Example**(p. 32) - How to initialise the library, create a multivector and compile a program.

Back to **Libcga Documentation**(p. **??**)

9.2 A Simple Example

The program `simpletest.c` is shown below. It is a simple program to create a multivector, map a 3D Euclidean vector and verifying that the result is a null-vector.

Libcga supports the `pkg-config` utility to provide the correct compiler and linker flags. To compile and run the program, change to the directory containing the source file and run the command

```
$ gcc -o simpletest simpletest.c `pkg-config libcga --cflags --libs`
```

If you get errors about libcga.pc not being found in the `pkg-config` search path, make sure the the directory containing the `libcga.pc` file installed with libcga is in the `PKG_CONFIG_PATH` environment variable.

Lets walk-through `simpletest.cc` line-by-line to explore what it does.

Firstly you must `#include` the appropriate header files containing the libcga functions you intend to use.

These pre-processor directives define some convenience macros to create/destroy multivector objects and getting/setting individual components. You should always access components in this way and never through the **components**(p. 23) field of multiv_t. This is to protect against the implementation changing in future releases of libcga.

Inside the main function we initialise the library. The call to **cga_init()**(p. 14) must be done before **any** other call to libcga. We also attempt to load the appropriate algebra, gracefully exiting if we fail for some reason (normally due to the correct plugin not being found).

We now create three multivector objects. Before you call **cga_new_multiv()**(p. 14) any multiv_t variables are invalid. To avoid memory leaks you should always call **cga_free_multiv()**(p. 15) on any creates multiv_t objects.

Using our convenience macros we set the e_1, e_2 and e_3 components of `a` so that it corresponds to the Euclidean vector (3, 4, 6).

cga_map We now use the **cga_map()**(p. 21) function to map `a` into the Conformal space. If we were dealing with hyperbolic space we could use the **cga_maph()**(p. 21) function. Note that these functions are defined in **cga_stdfunc.h**(p. 30) which is why we needed to `#include` that file.

The **cga_dump()**(p. 15) function dumps the contents of a multiv_t to stdout. Its mainly useful for debugging. At this point `a` should be a 5D vector.

The mapped vector should be a null-vector so we square `a` and dump the result to make sure it is zero. Note that we square `a` and store the result in `b` to make sure we don't overwrite the value of `a`.

Finally we inverse-map a and print the e_1, e_2 and e_3 components to verify that the inverse mapping works correctly.

Now we are finished with the multivectors we must free the resources they used.

Now we have finished using libcga we can call **cga_finalise()**(p. 14). No calls to libcga should be made after this call.

Back to **Examples of Libcga Use**(p. 31), **Libcga Documentation**(p. ??)

Back to **Examples of Libcga Use**(p. 31), **Libcga Documentation**(p. ??)

9.3 Introduction to Geometric Algebra

9.3.1 Introduction

Since its inception in the mid-1970s, Computer Graphics (CG) has almost universally used linear algebra as its mathematical framework. This may be due to two factors; most early practitioners of computer graphics were mathematicians familiar with it and linear algebra provided a compact, efficient way of representing points, transformations, lines, etc.

As computing power becomes cheaper, the opportunity arises to investigate new frameworks for CG which, although not providing the time/space efficiency of linear algebra, may provide a conceptually simpler system or one of greater analytical power.

9.3.1.1 Brief overview of Geometric Algebra

We shall assume a basic familiarity with Clifford algebras and merely describe the mechanism whereby they may be used to perform geometric operations.

We first write down a basis which can generate all elements of the Clifford algebra with vector elements in R^2 through linear combination

$$\{1, e_1, e_2, e_1e_2\}$$

where e_1, e_2 are the usual orthonormal Euclidean basis vectors and hence $e_1e_2 = e_1 \cdot e_2 + e_1 \wedge e_2 = e_1 \wedge e_2$. For convenience we shall denote e_ie_j as e_{ij} and, generally, $e_ie_j \ldots e_k$ as $e_{ij\ldots k}$. A general linear sum of these components is termed a *multivector* whereas a sum of only grade-n components is called a *n-vector*. This paper will use the convention of writing multivectors, bivectors, trivectors and higher-grade elements in upper-case and vectors and scalars in lower-case.

Firstly, note that

$$(e_1e_2)^2 = e_1e_2e_1e_2 = -e_1e_2e_2e_1 = -1$$

and thus we have an element of the algebra which squares to -1. As shown by Hestenes (1999) this can be identified with the unit imaginary $i = \sqrt{-1}$ in C and, as such, the highest-grade basis-element in a geometric algebra is often denoted I and referred to as the *pseudoscalar*.

Hestenes (1999) also showed that many of the identities and theorems dealing with complex numbers have a direct analogue in Clifford algebra and hence suggested a geometrical interpretation he named *Geometric Algebra*.

To demonstrate the approach, consider three orthonormal basis vectors in R^3, $\{e_1, e_2, e_3\}$. We can form 3 different bivectors from these vectors

$$B_1 = e_2e_3, \quad B_2 = e_3e_1, \quad B_3 = e_1e_2$$

Now consider the effect of B_3 upon the vectors e_1 and $e_1 + e_2$:

$$e_1B_3 = e_1e_1e_2 = e_1^2e_2 = e_2$$

$$(e_1 + e_2)B_3 = e_1B_3 + e_2B_3 = e_2 + e_2e_1e_2 = e_2 - e_1e_2^2 = e_2 - e_1$$

latex rotation.eps

The rotation effect upon the vectors e_1 and $e_1 + e_2$ produced by forming the Clifford product with the bivector $B_3 = e_1e_2$.

It is clear that B_3 has the effect of rotating the vectors by $-\pi/2$. It is, in fact, a general property that the bivector $e_i e_j$ will rotate a vector $-\pi/2$ in the plane defined by e_i and e_j. At first glance this seems to offer little more than complex numbers or quaternions but at no point have we assumed that we are working in 3-dimensional space; this method also extends to higher spaces.

Now we consider general rotations. Firstly it is trivial to show that B_3 squares to -1:

$$B_3^2 = e_1 e_2 e_1 e_2 = -e_1 e_2 e_2 e_1 = -1$$

and we can represent any vector Z in the plane defined by e_1 and e_2 using

$$Z = r(e_1 \cos\theta + e_2 \sin\theta)$$

$$Z = e_1 r(\cos\theta + B_3 \sin\theta)$$

where r is the distance of the point Z from the origin and θ is the angle Z makes with e_1. Since $B_3^2 = -1$, by considering the power series expansion of the exponential function, it can be shown that

$$e^{B_3\theta} = \cos\theta + B_3 \sin\theta$$

which is analogous to de Moivre's theorem for complex numbers.

We can thus represent any vector x which lies in the plane of the bivector B_3 by

$$x = e_1 r e^{B_3\theta} = r e_1 \cos\theta + r e_2 \sin\theta$$

Hence we can state that the rotation of a vector x which lies in the $e_1 e_2$ plane by ϕ radians is accomplished by

$$x \mapsto x e^{B_3\phi} = x(\cos\phi + B_3 \sin\phi)$$

This may readily be extended to higher dimensions and general planes. To rotate some vector x which lies in a plane spanned by the orthonormal vectors a and b by one right angle, simply form the bivector $B_3 = ab$ and continue as above.

latex rotation2.eps

Rotating vectors in arbitrary planes.

Careful consideration must be given to the case where the vector to be rotated, x does not lie on the plane of rotation. Firstly decompose the vector into a component which lies in the plane $x_\parallel$ and one normal to the plane $x_\perp$

$$x = x_\parallel + x_\perp$$

Now consider the effect of the following

$$e^{-\frac{\phi}{2}B_3} x e^{\frac{\phi}{2}B_3} = \left(\cos\frac{\phi}{2} - B_3 \sin\frac{\phi}{2}\right)(x_\parallel + x_\perp)\left(\cos\frac{\phi}{2} + B_3 \sin\frac{\phi}{2}\right)$$

$$e^{-\frac{\phi}{2}B_3} x e^{\frac{\phi}{2}B_3} = x_\parallel(\cos\phi + B_3 \sin\phi) + x_\perp$$

since bivectors anti-commute with vectors in their plane (e.g. $e_1(e_2 e_1) = -e_2 = -(e_2 e_1)e_1$) and commute with vectors normal to the plane (e.g.\ $e_1(e_2 e_3) = (e_2 e_3)e_1$). We have thus succeeded in rotating the component of the vector which lies in the plane without affecting the component normal to the plane, that is we have rotated the vector around an axis normal to the plane.

This leads to a general method of rotation in any plane; we form a bivector of the form $R = e^{-B\phi/2}$ for a given anticlockwise rotation ϕ in a plane specified by the bivector B and the transformation is given by

$$x \mapsto RxR^{-1}$$

We refer to these bivectors which have a rotational effect as *rotors*.

The computation of R^{-1} is rather difficult analytically (and indeed can require a full 2^n-dimension matrix inversion for a space of dimension n). To combat this we define the *reversion* of a n-vector $X = e_i e_j ... e_k$ as $\tilde{X} = e_k ... e_j e_i$, i.e. the literal reversion of the components. Since the reversion of $e_i e_j$ is $e_j e_i = -e_i e_j$ and by looking at the expression for R, it is clear that $\tilde{R} \equiv R^{-1}$ for rotors. Computing $\tilde{R}$ is easier since it involves only a sign change for some orthogonal elements of a multivector.

9.3.2 Conformal Geometric Algebra

This introduction shall restrict itself to three- and two-dimensional geometries but the techniques shown generalize easily to higher dimensions (something not always true with the usual vector algebra).

9.3.2.1 3D Euclidean geometry

We firstly extend an orthonormal basis of R^3 by adding two more orthogonal basis vectors e and $\bar{e}$:

$$\{e_1, e_2, e_3, e, \bar{e}\}$$

where $e^2 = 1$, $\bar{e}^2 = -1$ and $e \cdot \bar{e} = 0$. We shall term this extended algebra $\mathcal{A}(4,1)$ to show that 4 basis vectors square to $+1$ and one basis vector squares to -1. In addition we define the vectors n and $\bar{n}$

$$n = e + \bar{e}, \quad \bar{n} = e - \bar{e}$$

and direct substitution shows that they are null vectors, i.e. $n^2 = \bar{n}^2 = 0$.

The projection $x \in R^3 \mapsto \mathrm{F}(x) \in \mathcal{A}(4,1)$ is performed using the mapping introduced by Hestenes & Sobcyck (1984)

$$F(x) = \frac{1}{2}\left(x^2 n + 2x + \bar{n}\right)$$

where the factor of $1/2$ has been introduced to ensure that the mapping is normalized such that

$$F(x) \cdot n = -1$$

and by direct substitution and simplification it is easy to show that

$$[F(x)]^2 = 0 \quad \forall\ x \in R^3$$

At this point, notice that we can identify the origin with $\bar{n}$ since $F(0) = \bar{n}$ and that as $x \to \infty$, $F(x)$ becomes parallel to n.

Note that this mapping is dimensionally inconsistent, both n and $\bar{n}$ are dimensionless whereas x has dimensions of length. We introduce a fundamental unit length scale, λ, into the equation to make it dimensionally consistent

$$F(x) = \frac{1}{2\lambda^2}\left(x^2 n + 2\lambda x - \lambda^2 \bar{n}\right)$$

where λ is usually set to be unity. A geometric interpretation of λ will be discussed when dealing with non-Euclidean space.

As one may expect from their relation to complex numbers, there exists an element of the algebra which performs rotation in the plane. These pure-rotation rotors all have the form e^{-B} where B has only components of the form e_{ij} and $i, j \in \{1, 2, 3\}$.

A useful property of this mapping is that pure-rotation rotors retain their properties as can be shown by considering the effect of a rotor $R = \exp(\frac{\theta}{2}e_{ij})$, $i, j \in \{1, 2, 3\}$ upon $F(x)$. Setting $\lambda = 1$ for the moment,

$$RF(x)\tilde{R} = \frac{1}{2}R(x^2 n + 2x - \bar{n})\tilde{R}$$

$$RF(x)\tilde{R} = \frac{1}{2}\left(x^2 Rn\tilde{R} + 2RxR - R\bar{n}\tilde{R}\right)$$

$$RF(x)\tilde{R} = F(Rx\tilde{R})$$

since rotors leave n and $\bar{n}$ invariant and $(Rx\tilde{R})^2 = x^2$.

A similar approach used to derive the forms of a pure-rotation rotor allows us to derive a rotor $T_a = \exp(na/2)$ which has the effect of translating the vector x along a, i.e. $T_a(F(x))\tilde{T}_a = F(x+a)$.

It can also be shown that the rotor $D_\alpha = \exp(\alpha e\bar{e}/2)$ has the effect of dilating x by a factor of $e^{-\alpha}$, i.e. $D_\alpha F(x)\tilde{D}_\alpha \propto F(e^{-\alpha}x)$

Finally, inversions ($x \mapsto x^2/x$) may be represented as $F(x) \mapsto eF(x)e$. This will become particularly important when discussing non-Euclidean geometry. It is clear that using this mapping has provided a similar advantage to that of homogeneous co-ordinates, namely that many rigid body transforms are multiplicative and as such any rigid-body transform may be represented by a rotor. In addition, transforms followed by or preceded by a dilation or inversion may also be represented multiplicatively.

latex table.eps

Representations of various geometric objects.

We have shown how rigid body transformations may be performed on a vector x by operating on its null-vector representation $F(x)$. We may also ask what form the multivector M takes if the solutions of $F(x) \wedge M = 0$ lie on a circle, sphere, line or plane. It has been shown in Lasenby *et al.* (2002) that the form of M depends only on the null-vector representation of points which lie on the object. The form of M for various objects are summarized in Table tab:objects}. Note that if we identify the vector n with the point at infinity, a line is just a special case of a circle which passes through infinity and a plane is equally a special case of a sphere. It becomes convenient, therefore, to group planes and spheres into the collective group *generalized spheres} and, similarly, to define a {generalized circle* as either a circle or a line.

Suppose we have a generalized sphere which passes through the points $x_1, \ldots x_4$. Hence, for all points x on the object, $F(x) \wedge M = 0$ where

$$M = F(x_1) \wedge F(x_2) \wedge F(x_3) \wedge F(x_4)$$

Now consider the effect of a rotor R upon M

$$RM\tilde{R} = RF(x_1)\tilde{R} \wedge RF(x_2)\tilde{R} \wedge RF(x_3)\tilde{R} \wedge RF(x_4)\tilde{R}$$

Since $R(a \wedge b)\tilde{R} = R(a)\tilde{R} \wedge R(b)\tilde{R}$. Furthermore, for rigid-body transformation rotors, we can say

$$RM\tilde{R} = F(Rx_1\tilde{R}) \wedge F(Rx_2\tilde{R}) \wedge F(Rx_3\tilde{R}) \wedge F(Rx_4\tilde{R})$$

which represents a generalized sphere passing through the transformed points $X_n = Rx_n\tilde{R}$. Clearly, if R represents a conformal transformation, the generalized sphere represented by M is similarly transformed.

Intersections may be performed efficiently using the *meet* operator. The general meet operation can be found by noting that for r-grade and s-grade blades M_r and M_s, a point, X, on the intersection must satisfy $X \wedge M_r = X \wedge M_s = 0$ which can then be shown to be equivalent to

$$X \wedge \left[\langle M_r M_s \rangle_{2l-r-s}\right]^* = 0$$

where $[\cdot]^*$ denotes multiplication by the pseudoscalar, l is the dimension of the space (in the case of $A(4,1)$, $l = 5$) and $\langle X \rangle_i$ denotes the extraction of the i-grade component from X. If we define the meet operator

$$M_r \vee M_s = \left[\langle M_r M_s \rangle_{2l-r-s}\right]^*$$

Then we can interpret the meet of two objects as their intersection.

The key feature of this approach is that we have placed few constraints on the form of M_r and M_s and thus we can intersect objects in a fairly general manner instead of using object-specific algorithms.

Back to **Libcga Documentation**(p. ??)

9.4 Grade Tracking

Many existing implementations such as CLU and Gaigen represent the geometric product AB as a linear map, dependent on A, applied to the components of B. The matrix generally used to represent the mapping, A^G, depends only on the components of A. The vector-representation of the product, C, is found in the following manner,

$$\mathbf{C}{=}A^G\mathbf{B}$$

Using this method, in a typical case, extracting the centre, radius and normal vector of a circle passing through three points was found to require 109 floating-point operations (flops) using linear algebra and 3519 flops using the CGA approach.

Despite the intuitive nature of the CGA approach (for example finding intersections is dramatically easier when you have many different classes of object) it was almost 30 times slower than the classical approach.

A significant reduction in the operation count can be obtained by writing specialized product routines. For example the full geometric product calculation for $\mathcal{A}(4,1)$, with its 32 orthonormal basis-elements, requires a 32x32 matrix multiplication which requires 1024 floating-point multiplies. When you calculate the result of a bivector–bivector product, however, you need only perform 100 multiplies. The problem with this approach is that by having to use specialized routines, the generality CGA provides was lost. A solution used within libcga is to keep track of which grades are present in the multivector.

9.4.1 Grade tracking

The representation of a general multivector M is a sum of single-grade objects

$$M = \langle M\rangle_0 + \langle M\rangle_1 + ... + \langle M\rangle_n$$

and so the product of multivectors A and B is

$$AB = \langle A\rangle_0 \langle B\rangle_0 + \langle A\rangle_0 \langle B\rangle_1 + ... + \langle A\rangle_1 \langle B\rangle_0 + \langle A\rangle_1 \langle B\rangle_1 + ... + \langle A\rangle_n \langle B\rangle_{n-1} + \langle A\rangle_n \langle B\rangle_n$$

If G_A is the set of grades present in A and G_B is the set of grades present in B then

$$\langle A\rangle_i = 0 \quad \text{if } i \notin G_A$$

$$\langle B\rangle_i = 0 \quad \text{if } i \notin G_B$$

hence

$$\langle A\rangle_i \langle B\rangle_j = 0 \quad \text{if } i \notin G_A \text{ or } j \notin G_B$$

and need not be computed. If G_A and G_B are sufficiently small with respect to the dimension of the space, then significant advantage may be obtained. Since we always know a bivector–scalar product will return another bivector, the set of grades produced by the product may be computed directly from G_A and G_B. Hence, if the set G_M is kept with the representation of M we can always know which single-grade products will always return zero and thus need not be computed.

latex parallel.eps "" width=7cm

Block diagram showing how the product function may be parallelized.

The figure above shows this algorithm in block diagram form. Multivectors are broken down into single-grade components and those components which are zero are discarded. Each pair of components from each input multivector is passed to a specialized product routine and the results from each routine are concatenated. Since we discard some components prior to finding the products, some of the specialized routines will have no inputs and hence need not be calculated.

Back to **Libcga Documentation**(p. ??)

Index

acknowledgements, 2

cga.h, 27
cga_add
    core, 14
cga_assign_vector
    cga_stdfunc, 20
cga_cpy
    core, 15
cga_dilator
    cga_stdfunc, 21
cga_dual
    core, 16
cga_dump
    core, 14
cga_extract_grade
    core, 15
cga_factorise_bivector
    core, 16
cga_finalise
    core, 12
cga_flops
    core, 16
cga_free_multiv
    core, 13
cga_geometric
    core, 15
cga_get_element
    core, 13
cga_imap
    cga_stdfunc, 21
cga_imaph
    cga_stdfunc, 21
cga_init
    core, 12
cga_inner
    core, 15
cga_load_algebra
    core, 12
cga_mag2
    core, 16
cga_make_zero
    core, 17
cga_map
    cga_stdfunc, 20
cga_maph
    cga_stdfunc, 21
cga_matrix_to_action
    cga_rotor_interpolate, 18
cga_matrix_to_rotor
    cga_rotor_interpolate, 18
cga_meet
    core, 15
cga_new_multiv
    core, 13
cga_optimisation_flags
    core, 16
cga_outer
    core, 14
cga_reset_flops
    core, 16
cga_reverse
    core, 14
cga_rotor
    cga_stdfunc, 21
cga_rotor_exp
    cga_rotor_interpolate, 19
cga_rotor_interpolate
    cga_matrix_to_action, 18
    cga_matrix_to_rotor, 18
    cga_rotor_exp, 19
    cga_rotor_log, 19
    cga_rotor_to_matrix, 19
cga_rotor_interpolate.h, 29
cga_rotor_log
    cga_rotor_interpolate, 19
cga_rotor_to_matrix
    cga_rotor_interpolate, 19
cga_rotorh
    cga_stdfunc, 22
cga_scale
    core, 16
cga_set_optimisations
    core, 16
cga_sphere
    cga_stdfunc, 22
cga_stdfunc
    cga_assign_vector, 20
    cga_dilator, 21
    cga_imap, 21

cga_imaph, 21
cga_map, 20
cga_maph, 21
cga_rotor, 21
cga_rotorh, 22
cga_sphere, 22
cga_translator, 21
cga_translatorh, 22
cga_stdfunc.h, 30
cga_stdmv
core, 13
cga_subtract
core, 14
cga_translator
cga_stdfunc, 21
cga_translatorh
cga_stdfunc, 22
CGA_TYPE
core, 12
components
multiv_s, 23
core
cga_add, 14
cga_cpy, 15
cga_dual, 16
cga_dump, 14
cga_extract_grade, 15
cga_factorise_bivector, 16
cga_finalise, 12
cga_flops, 16
cga_free_multiv, 13
cga_geometric, 15
cga_get_element, 13
cga_init, 12
cga_inner, 15
cga_load_algebra, 12
cga_mag2, 16
cga_make_zero, 17
cga_meet, 15
cga_new_multiv, 13
cga_optimisation_flags, 16
cga_outer, 14
cga_reset_flops, 16
cga_reverse, 14
cga_scale, 16
cga_set_optimisations, 16
cga_stdmv, 13
cga_subtract, 14
CGA_TYPE, 12
multiv_t, 12
Core multivector manipulation functions, 11

dimension
multiv_s, 24

GPL, 2
grade_mask
multiv_s, 23

license, 2

max_grade
multiv_s, 23
multiv_s, 23
components, 23
dimension, 24
grade_mask, 23
max_grade, 23
num_components, 23
multiv_t
core, 12

num_components
multiv_s, 23

plugin_s, 25

Rotor interpolation functions for CGA, 18

Standard functions for CGA, 20